5 Creating a JSON API with minimal APIs

This chapter covers

- Creating a minimal API application to return JSON to clients
- Generating responses with IResult
- Using filters to perform common actions like validation
- Organizing your APIs with route groups

So far in this book you've seen several examples of minimal API applications that return simple `Hello World!` responses. These examples are great for getting started, but you can also use minimal APIs to build full-featured HTTP API applications. In this chapter you'll learn about HTTP APIs, see how they differ from a server-rendered application, and find out when to use them.

Section 5.2 starts by expanding on the minimal API applications you've already seen. You'll explore some basic routing concepts and show how values can be extracted from the URL automatically. Then you'll learn how to handle additional HTTP verbs such as `POST` and `PUT`, and explore various ways to define your APIs.

In section 5.3 you'll learn about the different return types you can use with minimal APIs. You'll see how to use the `Results` and `TypedResults` helper classes to easily create HTTP responses that use status codes like `201 Created` and `404 Not Found`. You'll also learn how to follow web standards for describing your errors by using the built-in support for Problem Details.

Section 5.4 introduces one of the big features added to minimal APIs in .NET 7: filters. You can use filters to build a mini pipeline (similar to the middleware pipeline from chapter 4) for each of your endpoints. Like middleware, filters are great for extracting common code from your endpoint handlers, making your handlers easier to read.

You'll learn about the other big .NET 7 feature for minimal APIs in section 5.5: *route groups*. You can use route groups to reduce the duplication in your minimal APIs, extracting common routing prefixes and filters, making your APIs easier to read, and reducing boilerplate. In conjunction with filters, route groups address many of the common complaints raised against minimal APIs when they were released in .NET 6.

One great aspect of ASP.NET Core is the variety of applications you can create with it. The ability to easily build a generalized HTTP API presents the possibility of using ASP.NET Core in a greater range of situations than can be achieved with traditional web apps alone. But *should* you build an HTTP API, and if so, why? In the first section of this chapter, I'll go over some of the reasons why you may—or may not—want to create a web API.

5.1 What is an HTTP API, and when should you use one?

Traditional web applications handle requests by returning HTML, which is displayed to the user in a web browser. You can easily build applications like that by using Razor Pages to generate HTML with Razor templates, as you'll learn in part 2 of this book. This approach is common and well understood, but the modern application developer has other possibilities to consider (figure 5.1), as you first saw in chapter 2.

Client-side single-page applications (SPAs) have become popular in recent years with the development of frameworks such as Angular, React, and Vue. These frameworks typically use JavaScript running in a web browser to generate the HTML that users see and interact with. The server sends this initial JavaScript to the browser when the user first reaches the app. The user's browser loads the JavaScript and initializes the SPA before loading any application data from the server.

> **NOTE** Blazor WebAssembly is an exciting new SPA framework. Blazor lets you write an SPA that runs in the browser like other SPAs, but it uses C# and Razor templates instead of JavaScript by using the new web standard, WebAssembly. I don't cover Blazor in this book, so to find out more, I recommend *Blazor in Action*, by Chris Sainty (Manning, 2022).

Once the SPA is loaded in the browser, communication with a server still occurs over HTTP, but instead of sending HTML directly to the browser in response to requests,

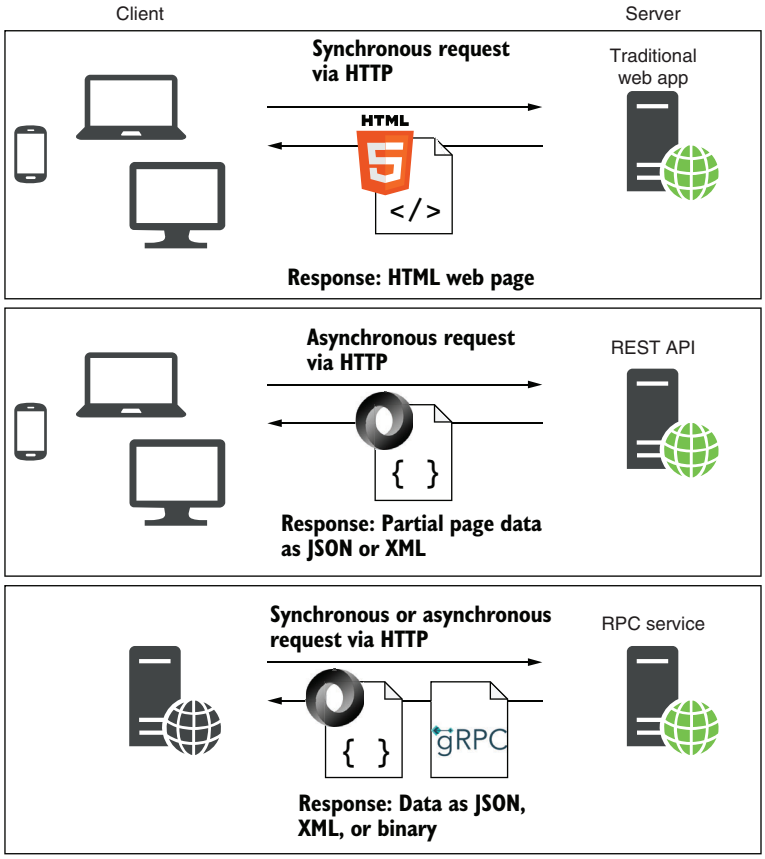


Figure 5.1 Modern developers have to consider several consumers of their applications. As well as traditional users with web browsers, these users could be single-page applications, mobile applications, or other apps.

the server-side application sends data—normally, in the ubiquitous JavaScript Object Notation (JSON) format—to the client-side application. Then the SPA parses the data and generates the appropriate HTML to show to a user, as shown in figure 5.2. The server-side application endpoint that the client communicates with is sometimes called an *HTTP API*, a *JSON API*, or a *REST API*, depending on the specifics of the API's design.

> **DEFINITION** An *HTTP API* exposes multiple URLs via HTTP that can be used to access or change data on a server. It typically returns data using the JSON format. HTTP APIs are sometimes called web APIs, but as *web API* refers to a specific technology in ASP.NET Core, in this book I use *HTTP API* to refer to the generic concept.

These days, mobile applications are common and, from the server application's point of view, similar to client-side SPAs. A mobile application typically communicates with a

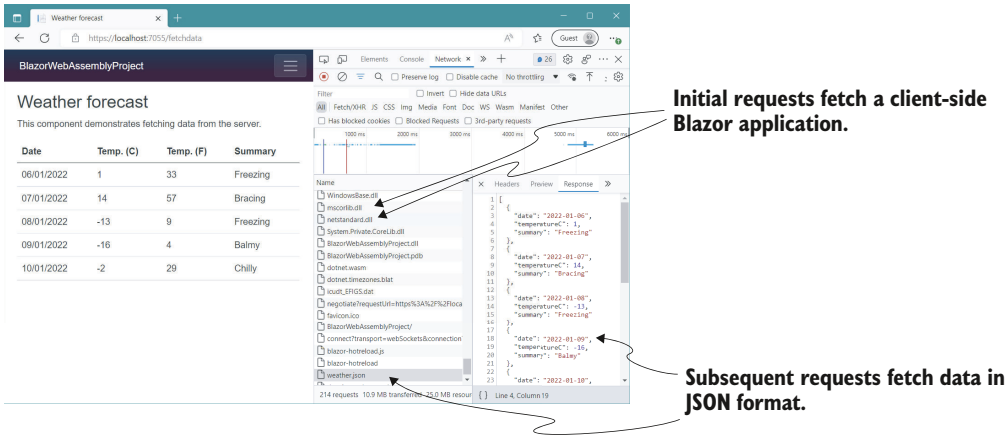


Figure 5.2 A sample client-side SPA using Blazor WebAssembly. The initial requests load the SPA files into the browser, and subsequent requests fetch data from a web API, formatted as JSON.

server application by using an HTTP API, receiving data in JSON format, just like an SPA. Then it modifies the application's UI depending on the data it receives.

One final use case for an HTTP API is where your application is designed to be partially or solely consumed by other backend services. Imagine that you've built a web application to send emails. By creating an HTTP API, you can allow other application developers to use your email service by sending you an email address and a message. Virtually all languages and platforms have access to an HTTP library they could use to access your service from code.

That's all there is to an HTTP API: it exposes endpoints (URLs) that client applications can send requests to and retrieve data from. These endpoints are used to power the behavior of the client apps, as well as to provide all the data the client apps need to display the correct interface to a user.

> **NOTE** You have even more options when it comes to creating APIs in ASP.NET Core. You can create remote procedure call APIs using gRPC, for example, or provide an alternative style of HTTP API using the GraphQL standard. I don't cover those technologies in this book, but you can read about gRPC at https://docs.microsoft.com/aspnet/core/grpc and find out about GraphQL in *Building Web APIs with ASP.NET Core,* by Valerio De Sanctis (Manning, 2023).

Whether you need or want to create an HTTP API for your ASP.NET Core application depends on the type of application you want to build. Perhaps you're familiar with client-side frameworks, or maybe you need to develop a mobile application, or you already have an SPA build pipeline configured. In each case, you'll most likely want to add HTTP APIs for the client apps to access your application.

One selling point for using an HTTP API is that it can serve as a generalized backend for all your client applications. You could start by building a client-side application that

uses an HTTP API. Later, you could add a mobile app that uses the same HTTP API, making little or no modification to your ASP.NET Core code.

If you're new to web development, HTTP APIs can also be easier to understand initially, as they typically return only JSON. Part 1 of this book focuses on minimal APIs so that you can focus on the mechanics of ASP.NET Core without needing to write HTML or CSS.

In part 3, you'll learn how to use Razor Pages to create server-rendered applications instead of minimal APIs. Server-rendered applications can be highly productive. They're generally recommended when you have no need to call your application from outside a web browser or when you don't want or need to make the effort of configuring a client-side application.

> **NOTE** Although there's been an industry shift toward client-side frameworks, server-side rendering using Razor is still relevant. Which approach you choose depends largely on your preference for building HTML applications in the traditional manner versus using JavaScript (or Blazor!) on the client.

Having said that, whether to use HTTP APIs in your application isn't something you necessarily have to worry about ahead of time. You can always add them to an ASP.NET Core app later in development, as the need arises.

SPAs with ASP.NET Core

The cross-platform, lightweight design of ASP.NET Core means that it lends itself well to acting as a backend for your SPA framework of choice. Given the focus of this book and the broad scope of SPAs in general, I won't be looking at Angular, React, or other SPAs here. Instead, I suggest checking out the resources appropriate to your chosen SPA. Books are available from Manning for all the common client-side JavaScript frameworks, as well as Blazor:

- *React in Action*, by Mark Tielens Thomas (Manning, 2018)
- *Angular in Action*, by Jeremy Wilken (Manning, 2018)
- *Vue.js in Action*, by Erik Hanchett with Benjamin Listwon (Manning, 2018)
- *Blazor in Action*, by Chris Sainty (Manning, 2022)

After you've established that you need an HTTP API for your application, creating one is easy, as it's the default application type in ASP.NET Core! In the next section we look at various ways you can create minimal API endpoints and ways to handle multiple HTTP verbs.

5.2 Defining minimal API endpoints

Chapters 3 and 4 gave you an introduction to basic minimal API endpoints. In this section, we'll build on those basic apps to show how you can handle multiple HTTP verbs and explore various ways to write your endpoint handlers.

5.2.1 Extracting values from the URL with routing

You've seen several minimal API applications in this book, but so far, all the examples have used fixed paths to define the APIs, as in this example:

```
app.MapGet("/", () => "Hello World!");
app.MapGet("/person", () => new Person("Andrew", "Lock");
```

These two APIs correspond to the paths `/` and `/person`, respectively. This basic functionality is useful, but typically you need some of your APIs to be more dynamic. It's unlikely, for example, that the `/person` API would be useful in practice, as it always returns the same `Person` object. What might be more useful is an API to which you can provide the user's first name, and the API returns all the users with that name.

You can achieve this goal by using *parameterized routes* for your API definitions. You can create a parameter in a minimal API route using the expression `{someValue}`, where `someValue` is any name you choose. The value will be extracted from the request URL's path and can be used in the lambda function endpoint.

> **NOTE** I introduce only the basics of extracting values from routes in this chapter. You'll learn a lot more about routing in chapter 6, including why we use routing and how it fits into the ASP.NET Core pipeline, as well as the syntax you can use.

If you create an API using the route template `/person/{name}`, for example, and send a request to the path `/person/Andrew`, the `name` parameter will have the value `"Andrew"`. You can use this feature to build more useful APIs, such as the one shown in the following listing.

Listing 5.1 A minimal API that uses a value from the URL

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var people = new List<Person>
{
    new("Tom", "Hanks"),
    new("Denzel", "Washington"),
    new("Leondardo", "DiCaprio"),
    new("Al", "Pacino"),
    new("Morgan", "Freeman"),
};

app.MapGet("/person/{name}", (string name) =>
    people.Where(p => p.FirstName.StartsWith(name)));

app.Run();
```

Creates a list of people as the data for the API

The route is parameterized to extract the name from the URL.

The extracted value can be injected into the lambda handler.

If you send a request to `/person/Al` for the app defined in listing 5.1, the `name` parameter will have the value `"Al"`, and the API will return the following JSON:

```
[{"firstName":"Al","lastName":"Pacino"}]
```

> **NOTE** By default, minimal APIs serialize C# objects to JSON. You'll see how to return other types of results in section 5.3.

The ASP.NET Core routing system is quite powerful, and we'll explore it in more detail in chapter 6. But with this simple capability, you can already build more complex applications.

5.2.2 Mapping verbs to endpoints

So far in this book we've defined all our minimal API endpoints by using the `MapGet()` function. This function matches requests that use the `GET` HTTP verb. `GET` is the most-used verb; it's what a browser uses when you enter a URL in the address bar of your browser or follow a link on a web page.

You should use `GET` only to *get* data from the server, however. You should never use it to *send* data or to *change* data on the server. Instead, you should use an HTTP verb such as `POST` or `DELETE`. You generally can't use these verbs by navigating web pages in the browser, but they're easy to send from a client-side SPA or mobile app.

> **TIP** If you're new to web programming or are looking for a refresher, Mozilla Developer Network (MDN), maker of the Firefox web browser, has a good introduction to HTTP at http://mng.bz/KeMK.

In theory, each of the HTTP verbs has a well-defined purpose, but in practice, you may see apps that only ever use `POST` and `GET`. This is often fine for server-rendered applications like Razor Pages, as it's typically simpler, but if you're creating an API, I recommend that you use the HTTP verbs with the appropriate semantics wherever possible.

You can define endpoints for other verbs with minimal APIs by using the appropriate `Map*` functions. To map a `POST` endpoint, for example, you'd use `MapPost()`. Table 5.1 shows the minimal API `Map*` methods available, the corresponding HTTP verbs, and the typical semantic expectations of each verb on the types of operations that the API performs.

Table 5.1 The minimal API map endpoints and the corresponding HTML verbs

Method	HTTP verb	Expected operation
`MapGet(path, handler)`	`GET`	Fetch data only; no modification of state. May be safe to cache.
`MapPost(path, handler)`	`POST`	Create a new resource.
`MapPut(path, handler)`	`PUT`	Create or replace an existing resource.

Table 5.1 The minimal API map endpoints and the corresponding HTML verbs *(continued)*

Method	HTTP verb	Expected operation
`MapDelete(path, handler)`	`DELETE`	Delete the given resource.
`MapPatch(path, handler)`	`PATCH`	Modify the given resource.
`MapMethods(path, methods, handler)`	Multiple verbs	Multiple operations.
`Map(path, handler)`	All verbs	Multiple operations.
`MapFallback(handler)`	All verbs	Useful for SPA fallback routes.

RESTful applications (as described in chapter 2) typically stick close to these verb uses where possible, but some of the actual implementations can differ, and people can easily get caught up in pedantry. Generally, if you stick to the expected operations described in table 5.1, you'll create a more understandable interface for consumers of the API.

> **NOTE** You may notice that if you use the `MapMethods()` and `Map()` methods listed in table 5.1, your API probably doesn't correspond to the expected operations of the HTTP verbs it supports, so I avoid these methods where possible. `MapFallback()` doesn't have a path and is called *only* if no other endpoint matches. Fallback routes can be useful when you have a SPA that uses client-side routing. See http://mng.bz/9DMl for a description of the problem and an alternative solution.

As I mentioned at the start of section 5.2.2, testing APIs that use verbs other than `GET` is tricky in the browser. You need to use a tool that allows sending arbitrary requests such as Postman (https://www.postman.com) or the HTTP Client plugin in JetBrains Rider. In chapter 11 you'll learn how to use a tool called Swagger UI to visualize and test your APIs.

> **TIP** The HTTP client plugin in JetBrains Rider makes it easy to craft HTTP requests from inside your API, and even discovers all the endpoints in your application automatically, making them easier to test. You can read more about it at https://www.jetbrains.com/help/rider/Http_client_in__product__code_editor.html.

As a final note before we move on, it's worth mentioning the behavior you get when you call a method with the *wrong* HTTP verb. If you define an API like the one in listing 5.1

```
app.MapGet("/person/{name}", (string name) =>
  people.Where(p => p.FirstName.StartsWith(name)));
```

and call it by using a `POST` request to `/person/Al` instead of a `GET` request, the handler won't run, and the response you get will have status code `405 Method Not Allowed`.

TIP You should never see this response when you're calling the API correctly, so if you receive a `405` response, make sure to check that you're using the right HTTP verb and the right path. Often when I see a `405`, I've used the correct verb but made a typo in the URL!

In all the examples in this book so far, you provide a lambda function as the handler for an endpoint. But in section 5.2.3, you'll see that there are many ways to define the handler.

5.2.3 Defining route handlers with functions

For basic examples, using a lambda function as the handler for an endpoint is often the simplest approach, but you can take many approaches, as shown in the following listing. This listing also demonstrates creating a simple CRUD (Create, Read, Update, Delete) API using different HTTP verbs, as discussed in section 5.2.1.

Listing 5.2 Creating route handlers for a simple CRUD API

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/fruit", () => Fruit.All);

var getFruit = (string id) => Fruit.All[id];
app.MapGet("/fruit/{id}", getFruit);

app.MapPost("/fruit/{id}", Handlers.AddFruit);

Handlers handlers = new();
app.MapPut("/fruit/{id}", handlers.ReplaceFruit);

app.MapDelete("/fruit/{id}", DeleteFruit);

app.Run();

void DeleteFruit(string id)
{
    Fruit.All.Remove(id);
}

record Fruit(string Name, int Stock)
{
    public static readonly Dictionary<string, Fruit> All = new();
};

class Handlers
{
    public void ReplaceFruit(string id, Fruit fruit)
    {
        Fruit.All[id] = fruit;
    }
```

Lambda expressions are the simplest but least descriptive way to create a handler.

Storing the lambda expression as a variable means you can name it—getFruit in this case.

Handlers can be static methods in any class.

Handlers can also be instance methods.

You can also use local functions, introduced in C# 7.0, as handler methods.

Handlers can also be instance methods.

```
    public static void AddFruit(string id, Fruit fruit)
    {
        Fruit.All.Add(id, fruit);
    }
}
```

Converts the response to a JsonObject

Listing 5.2 demonstrates the various ways you can pass handlers to an endpoint by simulating a simple API for interacting with a collection of Fruit items:

- A lambda expression, as in the `MapGet("/fruit")` endpoint
- A `Func<T, TResult>` variable, as in the `MapGet("/fruit/{id}")` endpoint
- A static method, as in the `MapPost` endpoint
- A method on an instance variable, as in the `MapPut` endpoint
- A local function, as in the `MapDelete` endpoint

All these approaches are functionally identical, so you can use whichever pattern works best for you.

Each `Fruit` record in listing 5.2 has a `Name` and a `Stock` level and is stored in a dictionary with an `id`. You call the API by using different HTTP verbs to perform the CRUD operations against the dictionary.

> **WARNING** This API is simple. It isn't thread-safe, doesn't validate user input, and doesn't handle edge cases. We'll remedy some of those deficiencies in section 5.3.

The handlers for the `POST` and `PUT` endpoints in listing 5.2 accept both an `id` parameter and a `Fruit` parameter, showing another important feature of minimal APIs. *Complex types*—that is, types that can't be extracted from the URL by means of route parameters—are created by deserializing the JSON body of a request.

> **NOTE** By contrast with APIs built using ASP.NET and ASP.NET Core web API controllers (which we cover in chapter 20), minimal APIs can bind only to JSON bodies and always use the System.Text.Json library for JSON deserialization.

Figure 5.3 shows an example of a `POST` request sent with Postman. Postman sends the request body as JSON, which the minimal API automatically deserializes into a `Fruit` instance before calling the endpoint handler. You can bind only a single object in your endpoint handler to the request body in this way. I cover model binding in detail in chapter 7.

Minimal APIs leave you free to organize your endpoints any way you choose. That flexibility is often cited as a reason to *not* use them, due to the fear that developers will keep all the functionality in a single file, as in most examples (such as listing 5.2). In practice, you'll likely want to extract your endpoints to separate files so as to modularize them and make them easier to understand. Embrace that urge; that's the way they were intended to be used!

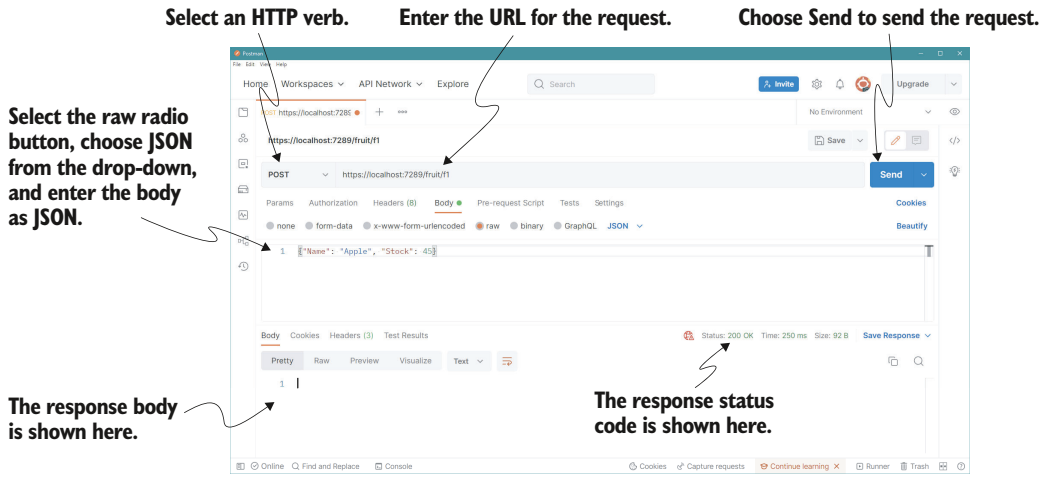


Figure 5.3 Sending a `POST` request with Postman. The minimal API automatically deserializes the JSON in the request body to a `Fruit` instance before calling the endpoint handler.

Now you have a simple API, but if you try it out, you'll quickly run into scenarios in which your API seems to break. In section 5.3 you learn how to handle some of these scenarios by returning status codes.

5.3 Generating responses with IResult

You've seen the basics of minimal APIs, but so far, we've looked only at the happy path, where you can handle the request successfully and return a response. In this section we look at how to handle bad requests and other errors by returning different status codes from your API.

The API in listing 5.2 works well as long as you perform only operations that are valid for the current state of the application. If you send a `GET` request to `/fruit`, for example, you'll always get a `200` success response, but if you send a `GET` request to `/fruit/f1` *before* you create a `Fruit` with the `id f1`, you'll get an exception and a `500 Internal Server Error` response, as shown in figure 5.4.

Throwing an exception whenever a user requests an `id` that doesn't exist clearly makes for a poor experience all round. A better approach is to return a status code indicating the problem, such as `404 Not Found` or `400 Bad Request`. The most declarative way to do this with minimal APIs is to return an `IResult` instance.

All the endpoint handlers you've seen so far in this book have returned `void`, a `string`, or a plain old CLR object (POCO) such as `Person` or `Fruit`. There is one other type of object you can return from an endpoint: an `IResult` implementation.

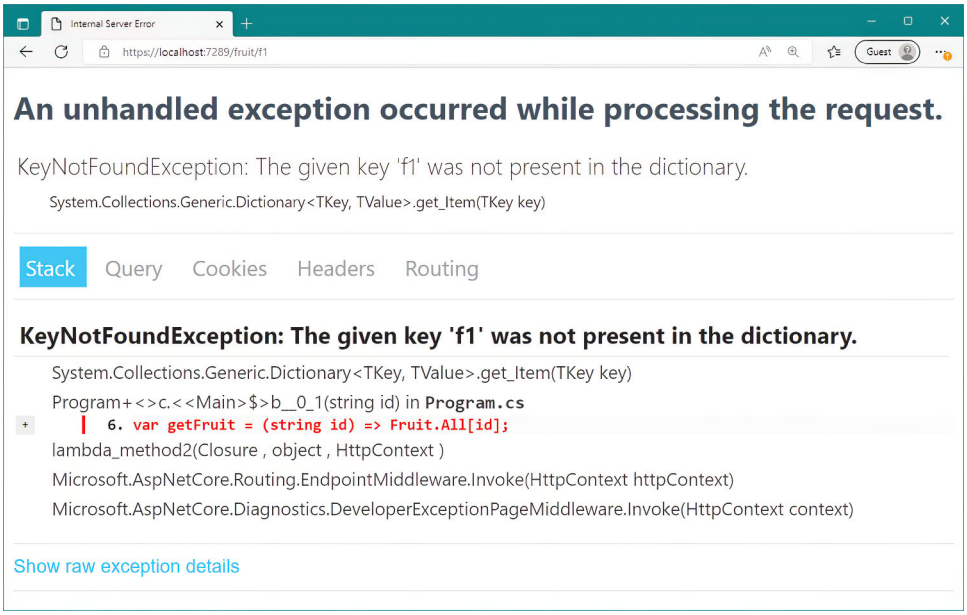


Figure 5.4 If you try to retrieve a fruit by using a nonexistent `id` for the simplistic API in listing 5.2, the endpoint throws an exception. This exception is handled by the `DeveloperExceptionPageMiddleware` but provides a poor experience.

In summary, the endpoint middleware handles each return type as follows:

- `void` or `Task`—The endpoint returns a `200` response with no body.
- `string` or `Task<string>`—The endpoint returns a `200` response with the string serialized to the body as `text/plain`.
- `IResult` or `Task<IResult>`—The endpoint executes the `IResult.ExecuteAsync` method. Depending on the implementation, this type can customize the response, returning any status code.
- `T` or `Task<T>`—All other types (such as POCO objects) are serialized to JSON and returned in the body of a `200` response as `application/json`.

The `IResult` implementations provide much of the flexibility in minimal APIs, as you'll see in section 5.3.1.

5.3.1 Returning status codes with Results and TypedResults

A well-designed API uses status codes to indicate to a client what went wrong when a request failed, as well as potentially provide more descriptive codes when a request is successful. You should anticipate common problems that may occur when clients call your API and return appropriate status codes to indicate the causes to users.

ASP.NET Core exposes the simple `static` helper types `Results` and `TypedResults` in the namespace `Microsoft.AspNetCore.Http`. You can use these helpers to create a response with common status codes, optionally including a JSON body. Each of the

methods on `Results` and `TypedResults` returns an implementation of `IResult`, which the endpoint middleware executes to generate the final response.

> **NOTE** `Results` and `TypedResults` perform the same function, as helpers for generating common status codes. The only difference is that the `Results` methods return an `IResult`, whereas `TypedResults` return a concrete generic type, such as `Ok<T>`. There's no difference in terms of functionality, but the generic types are easier to use in unit tests and in OpenAPI documentation, as you'll see in chapters 36 and 11. `TypedResults` were added in .NET 7.

The following listing shows an updated version of listing 5.2, in which we address some of the deficiencies in the API and use `Results` and `TypedResults` to return different status codes to clients.

Listing 5.3 Using `Results` and `TypedResults` in a minimal API

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit", () => _fruit);

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.NotFound());

app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.BadRequest(new
            { id = "A fruit with this id already exists" }));

app.MapPut("/fruit/{id}", (string id, Fruit fruit) =>
{
    _fruit[id] = fruit;
    return Results.NoContent();
});

app.MapDelete("/fruit/{id}", (string id) =>
{
    _fruit.TryRemove(id, out _);
    return Results.NoContent();
});

app.Run();
record Fruit(string Name, int stock);
```

Uses a concurrent dictionary to make the API thread-safe

Tries to get the fruit from the dictionary. If the ID exists in the dictionary, this returns true . . .

. . . and we return a 200 OK response, serializing the fruit in the body as JSON.

If the ID doesn't exist, returns a 404 Not Found response

Tries to add the fruit to the dictionary. If the ID hasn't been added yet. this returns true . . .

. . . and we return a 201 response with a JSON body and set the Location header to the given path.

If the ID already exists, returns a 400 Bad Request response with an error message

After adding or replacing the fruit, returns a 204 No Content response

After deleting the fruit, always returns a 204 No Content response

Listing 5.3 demonstrates several status codes, some of which you may not be familiar with:

- `200 OK`—The standard successful response. It often includes content in the body of the response but doesn't have to.
- `201 Created`—Often returned when you successfully created an entity on the server. The `Created` result in listing 5.3 also includes a `Location` header to describe the URL where the entity can be found, as well as the JSON entity itself in the body of the response.
- `204 No Content`—Similar to a 200 response but without any content in the response body.
- `400 Bad Request`—Indicates that the request was invalid in some way; often used to indicate data validation failures.
- `404 Not Found`—Indicates that the requested entity could not be found.

These status codes more accurately describe your API and can make an API easier to use. That said, if you use only `200 OK` responses for all your successful responses, few people will mind or think less of you! You can see a summary of all the possible status codes and their expected uses at http://mng.bz/jP4x.

> **NOTE** The `404` status code in particular causes endless debate in online forums. Should it be *only* used if the request didn't match an endpoint? Is it OK to use `404` to indicate a missing entity (as in the previous example)? There are endless proponents in both camps, so take your pick!

`Results` and `TypedResults` include methods for all the common status code results you could need, but if you don't want to use them for some reason, you can always set the status code yourself directly on the `HttpResponse`, as in listing 5.4. In fact, the listing shows how to define the entire response manually, including the status code, the content type, and the response body. You won't need to take this manual approach often, but it can be useful in some situations.

Listing 5.4 Writing the response manually using `HttpResponse`

```
using System.Net.Mime
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/teapot", (HttpResponse response) =>
{
    response.StatusCode = 418;
    response.ContentType = MediaTypeNames.Text.Plain;
    return response.WriteAsync("I'm a teapot!");
});

app.Run();
```

Accesses the HttpResponse by including it as a parameter in your endpoint handler

You can set the status code directly on the response.

Defines the content type that will be sent in the response

You can write data to the response stream manually.

`HttpResponse` represents the response that will be sent to the client and is one of the special types that minimal APIs know to inject into your endpoint handlers (instead of trying to create it by deserializing from the request body). You'll learn about the other types you can use in your endpoint handlers in chapter 7.

5.3.2 Returning useful errors with Problem Details

In the `MapPost` endpoint of listing 5.3, we checked to see whether an entity with the given `id` already existed. If it did, we returned a `400` response with a description of the error. The problem with this approach is that the client—typically, a mobile app or SPA—must know how to read and parse that response. If each of your APIs has a different format for errors, that arrangement can make for a confusing API. Luckily, a web standard called Problem Details describes a consistent format to use.

> **DEFINITION** Problem Details is a web specification (https://www.rfc-editor.org/rfc/rfc7807.html) for providing machine-readable errors for HTTP APIs. It defines the required and optional fields that should be in the JSON body for errors.

ASP.NET Core includes two helper methods for generating Problem Details responses from minimal APIs: `Results.Problem()` and `Results.ValidationProblem()` (plus their `TypedResults` counterparts). Both of these methods return Problem Details JSON. The only difference is that `Problem()` defaults to a `500` status code, whereas `ValidationProblem()` defaults to a `400` status and requires you to pass in a `Dictionary` of validation errors, as shown in the following listing.

Listing 5.5 Returning Problem Details using `Results.Problem`

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit", () => _fruit);

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404));

app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>
          {
              {"id", new[] {"A fruit with this id already exists"}}
          }));
```

Returns a Problem Details object with a 404 status code

Returns a Problem Details object with a 400 status code and includes the validation errors

The `ProblemHttpResult` returned by these methods takes care of including the correct title and description based on the status code, and generates the appropriate JSON, as shown in figure 5.5. You can override the default title and description by passing additional arguments to the `Problem()` and `ValidationProblem()` methods.

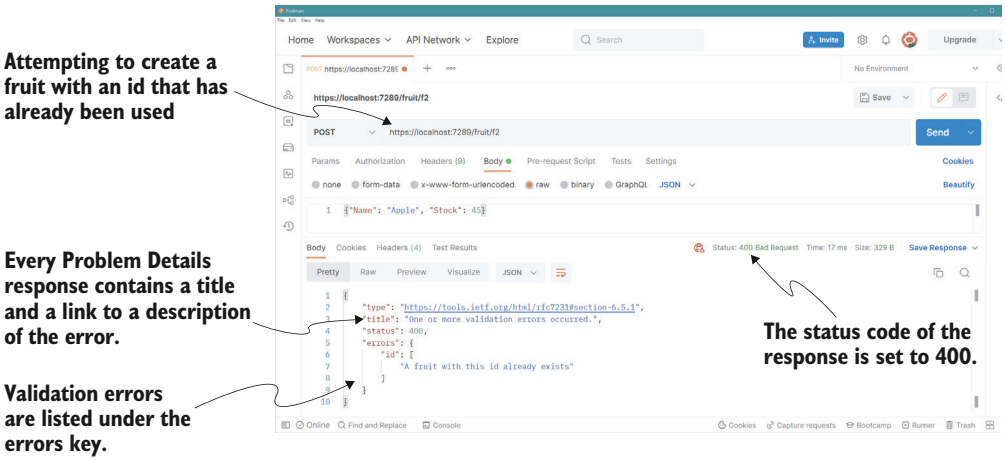


Figure 5.5 You can return a Problem Details response by using the `Problem` and `ValidationProblem` methods. The `ValidationProblem` response shown here includes a description of the error, along with the validation errors in a standard format. This example shows the response when you try to create a fruit with an `id` that has already been used.

Deciding on an error format is an important step whenever you create an API, and as Problem Details is already a web standard, it should be your go-to approach, especially for validation errors. Next, you'll learn how to ensure that all your error responses are Problem Details.

5.3.3 Converting all your responses to Problem Details

In section 5.3.2 you saw how to use the `Results.Problem()` and `Results.ValidationProblem()` methods in your minimal API endpoints to return Problem Details JSON. The only catch is that your minimal API endpoints aren't the *only* thing that could generate errors. In this section you'll learn how to make sure that all your errors return Problem Details JSON, keeping the error responses consistent across your application.

A minimal API application could generate an error response in several ways:

- Returning an error status code from an endpoint handler
- Throwing an exception in an endpoint handler, which is caught by the `ExceptionHandlerMiddleware` or the `DeveloperExceptionPageMiddleware` and converted to an error response
- The middleware pipeline returning a `404` response because a request isn't handled by an endpoint

- A middleware component in the pipeline throwing an exception
- A middleware component returning an error response because a request requires authentication, and no credentials were provided

There are essentially two classes of errors, which are handled differently: exceptions and error status code responses. To create a consistent API for consumers, we need to make sure that both error types return Problem Details JSON in the response.

CONVERTING EXCEPTIONS TO PROBLEM DETAILS

In chapter 4 you learned how to handle exceptions with the `ExceptionHandlerMiddleware`. You saw that the middleware catches any exceptions from later middleware and generates an error response by executing an error-handling path. You could add the middleware to your pipeline with an error-handling path of `"/error"`:

```
app.UseExceptionHandler("/error");
```

`ExceptionHandlerMiddleware` invokes this path after it captures an exception to generate the final response. The trouble with this approach for minimal APIs is that you need a dedicated error endpoint, the sole purpose of which is to generate a Problem Details response.

Luckily, in .NET 7, you can configure the `ExceptionHandlerMiddleware` (and `DeveloperExceptionPageMiddleware`) to convert an exception to a Problem Details response automatically. In .NET 7, you can add the new `IProblemDetailsService` to your app by calling `AddProblemDetails()` on `WebApplicationBuilder.Services`. When the `ExceptionHandlerMiddleware` is configured *without* an error-handling path, it automatically uses the `IProblemDetailsService` to generate the response, as shown in figure 5.6.

> **WARNING** Calling `AddProblemDetails()` registers the `IProblemDetailsService` service in the dependency injection container so that other services and middleware can use it. If you configure `ExceptionHandlerMiddleware` without an error-handling path but forget to call `AddProblemDetails()`, you'll get an exception when your app starts. You'll learn more about dependency injection in chapters 8 and 9.

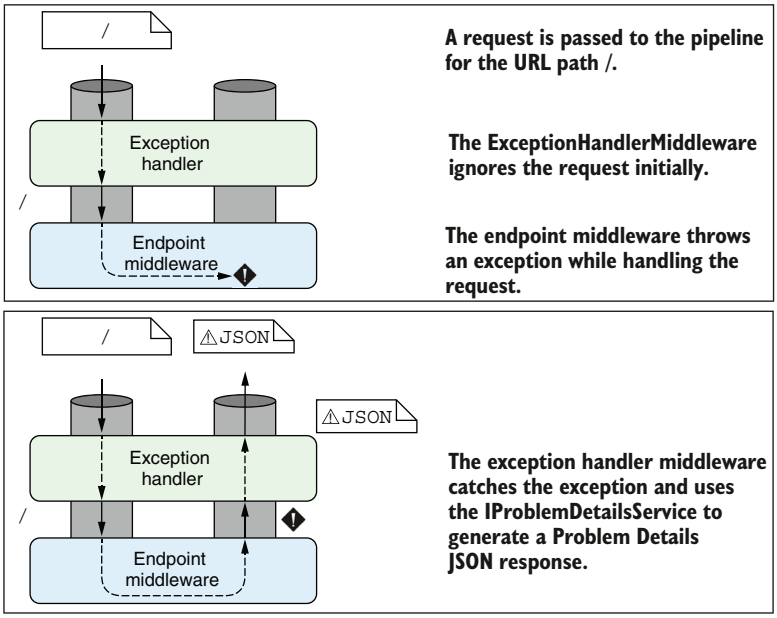


Figure 5.6 The `ExceptionHandlerMiddleware` catches exceptions that occur later in the middleware pipeline. If the middleware isn't configured to reexecute the pipeline, it generates a Problem Details response by using the `IProblemDetailsService`.

Listing 5.6 shows how to configure Problem Details generation in your exception handlers. Add the required `IProblemDetailsService` service to your app, and call `UseExceptionHandler()` without providing an error-handling path, and the middleware will generate a Problem Details response automatically when it catches an exception.

Listing 5.6 Configuring `ExceptionHandlerMiddleware` to use Problem Details

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();

WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();
}

app.MapGet("/", void () => throw new Exception());

app.Run();
```

Adds the IProblemDetailsService implementation

Configures the ExceptionHandlerMiddleware without a path so that it uses the IProblemDetailsService

Throws an exception to demonstrate the behavior

As discussed in chapter 4, `WebApplication` automatically adds the `DeveloperExceptionPageMiddleware` to your app in the development environment. This middleware similarly supports returning Problem Details when two conditions are satisfied:

- You've registered an `IProblemDetailsService` with the app (by calling `AddProblemDetails()` in Program.cs).
- The request indicates that it doesn't support HTML. If the client supports HTML, middleware uses the HTML developer exception page from chapter 4 instead.

The `ExceptionHandlerMiddleware` and `DeveloperExceptionPageMiddleware` take care of converting all your exceptions to Problem Details responses, but you still need to think about nonexception errors, such as the automatic `404` response generated when a request doesn't match any endpoints.

CONVERTING ERROR STATUS CODES TO PROBLEM DETAILS

Returning error status codes is the common way to communicate errors to a client with minimal APIs. To ensure a consistent API for consumers, you should return a Problem Details response whenever you return an error. Unfortunately, as already mentioned, you don't control all the places where an error code may be created. The middleware pipeline automatically returns a `404` response when an unmatched request reaches the end of the pipeline, for example.

Instead of generating a Problem Details response in your endpoint handlers, you can add middleware to convert responses to Problem Details automatically by using the `StatusCodePagesMiddleware`, as shown in figure 5.7. Any response that reaches the middleware with an error status code and doesn't already have a body has a Problem

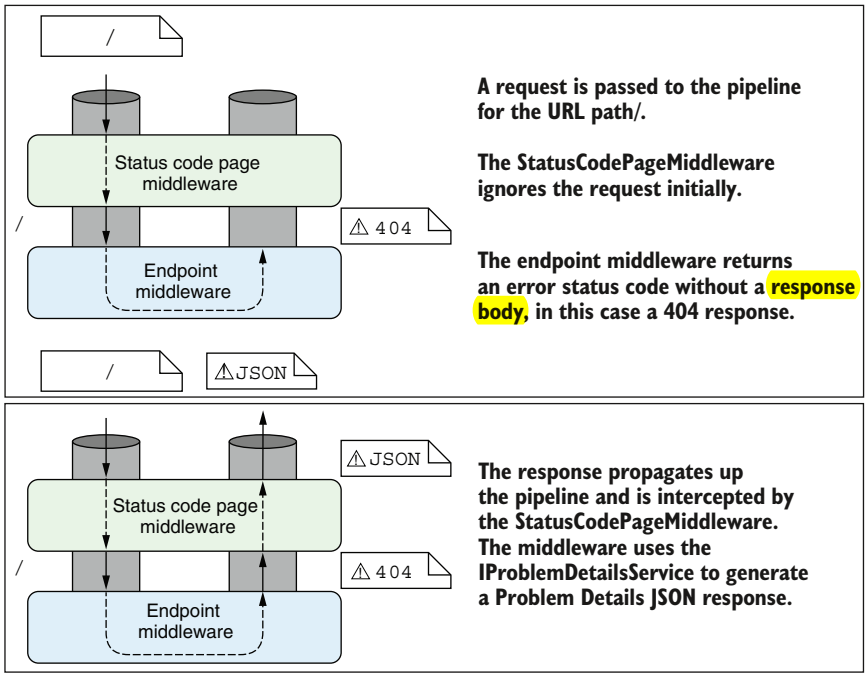


Figure 5.7 The `StatusCodePagesMiddleware` intercepts responses with an error status code that have no response body and adds a Problem Details response body.

Details body added by the middleware. The middleware converts all error responses automatically, regardless of whether they were generated by an endpoint or from other middleware.

> **NOTE** You can also use the `StatusCodePagesMiddleware` to reexecute the middleware pipeline with an error handling path, as you can with the `ExceptionHandlerMiddleware` (chapter 4). This technique is most useful for Razor Pages applications when you want to have a different error page for specific status codes, as you'll see in chapter 15.

Add the `StatusCodePagesMiddleware` to your app by using the `UseStatusCodePages()` extension method, as shown in the following listing. Ensure that you also add the `IProblemDetailsService` to your app by using `AddProblemDetails()`.

Listing 5.7 Using `StatusCodePagesMiddleware` to return Problem Details

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();

WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();
}

app.UseStatusCodePages();

app.MapGet("/", () => Results.NotFound());

app.Run();
```

Adds the IProblemDetailsService implementation

Adds the StatusCodePagesMiddleware

The StatusCodePagesMiddleware automatically adds a Problem Details body to the 404 response.

The `StatusCodePagesMiddleware`, coupled with exception-handling middleware, ensures that your API returns a Problem Details response for all error responses.

> **TIP** You can also customize how the Problem Details response is generated by passing parameters to the `AddProblemDetails()` method or by implementing your own `IProblemDetailsService`.

So far in section 5.3, I've described returning objects as JSON, returning a `string` as text, and returning custom status codes and Problem Details by using `Results`. Sometimes, however, you need to return something bigger, such as a file or a binary. Luckily, you can use the convenient `Results` class for that task too.

5.3.4 Returning other data types

The methods on `Results` and `TypedResults` are convenient ways of returning common responses, so it's only natural that they include helpers for other common scenarios, such as returning a file or binary data:

- `Results.File()`—Pass in the path of the file to return, and ASP.NET Core takes care of streaming it to the client.
- `Results.Byte()`—For returning binary data, you can pass this method a `byte[]` to return.
- `Results.Stream()`—You can send data to the client asynchronously by using a `Stream`.

In each of these cases, you can provide a content type for the data, and a filename to be used by the client. Browsers offer to save binary data files using the suggested filename. The `File` and `Byte` methods even support range requests by specifying `enableRangeProcessing` as `true`.

> **DEFINITION** Clients can create range requests using the `Range` header to request a specific range of bytes from the server instead of the whole file, reducing the bandwidth required for a request. When range requests are enabled for `Results.File()` or `Results.Byte()`, ASP.NET Core automatically handles generating an appropriate response. You can read more about range requests at http://mng.bz/Wzd0.

If the built-in `Results` helpers don't provide the functionality you need, you can always fall back to creating a response manually, as in listing 5.4. If you find yourself creating the same manual response several times, you could consider creating a custom `IResult` type to encapsulate this logic. I show how to create a custom `IResult` that returns XML and registers it as an extension in this blog post: http://mng.bz/8rNP.

5.4 Running common code with endpoint filters

In section 5.3 you learned how to use `Results` to return different responses when the request isn't valid. We'll look at validation in more detail in chapter 7, but in this section, you'll learn how to use filters to extract common code that executes before (or after) an endpoint executes.

Let's start by adding some extra validation to the `fruit` API from listing 5.5. The following listing adds an additional check to the `MapGet` endpoint to ensure that the provided `id` isn't empty and that it starts with the letter `f`.

Listing 5.8 Adding basic validation to minimal API endpoints

```
using System.Collections.Concurrent;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
{
    if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
    {
        return Results.ValidationProblem(new Dictionary<string, string[]>
```

Adds extra validation that the provided id has the required format

```
        {
            {"id", new[] {"Invalid format. Id must start with 'f'"}}
        });
    }

    return _fruit.TryGetValue(id, out var fruit)
            ? TypedResults.Ok(fruit)
            : Results.Problem(statusCode: 404);
});

app.Run()
```

Even though this check is basic, it starts to clutter our endpoint handler, making it harder to read what the endpoint is doing. One improvement would be to move the validation code to a helper function. But you're still inevitably going to clutter your endpoint handlers with calls to methods that are tangential to the main function of your endpoint.

NOTE Chapter 7 discusses additional validation patterns in detail.

It's common to perform various cross-cutting activities for every endpoint. I've already mentioned validation; other cross-cutting activities include logging, authorization, and auditing. ASP.NET Core has built-in support for some of these features, such as authorization (chapter 24), but you're likely to have some common code that doesn't fit into the specific pigeonholes of validation or authorization.

Luckily, ASP.NET Core includes a feature in minimal APIs for running these tangential concerns: endpoint filters. You can specify a filter for an endpoint by calling `AddEndpointFilter()` on the result of a call to `MapGet` (or similar) and passing in a function to execute. You can even add multiple calls to `AddEndpointFilter()`, which builds up an *endpoint filter pipeline*, analogous to the middleware pipeline. Figure 5.8 shows that the pipeline is functionally identical to the middleware pipeline in figure 4.3.

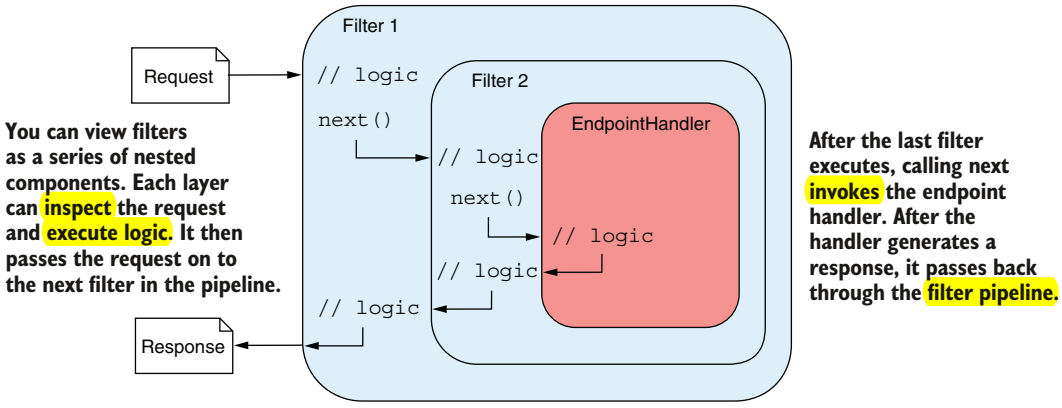


Figure 5.8 The endpoint filter pipeline. Filters execute code and then call `next(context)` to invoke the next filter in the pipeline. If there are no more filters in the pipeline, the endpoint handler is invoked. After the handler has executed, the filters may run further code.

Each endpoint filter has two parameters: a `context` parameter, which provides details about the selected endpoint handler, and the `next` parameter, which represents the filter pipeline. When you invoke the methodlike `next` parameter by calling `next(context)`, you invoke the remainder of the filter pipeline. If there are no more filters in the pipeline, you invoke the endpoint handler, as shown in figure 5.8.

Listing 5.9 shows how to run the same validation logic you saw in listing 5.8 in an endpoint filter. The filter function accesses the endpoint method arguments by using the `context.GetArgument<T>()` function, passing in a position; `0` is the first argument of your endpoint handler, `1` is the second argument, and so on. If the argument isn't valid, the filter function returns an `IResult` object response. If the argument is valid, the filter calls `await next(context)` instead, executing the endpoint handler.

Listing 5.9 Using `AddEndpointFilter` to extract common code

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter(ValidationHelper.ValidateId);

app.Run();

class ValidationHelper
{
    internal static async ValueTask<object?> ValidateId(
        EndpointFilterInvocationContext context,
        EndpointFilterDelegate next)
    {
        var id = context.GetArgument<string>(0);
        if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
        {
            return Results.ValidationProblem(
                new Dictionary<string, string[]>
                {
                    {"id", new[]{"Invalid format. Id must start with 'f'"}}
                });
        }

        return await next(context);
    }
}
```

Adds the filter to the endpoint using AddEndpointFilter

context exposes the endpoint method arguments and the HttpContext.

The method must return a ValueTask.

next represents the filter method (or endpoint) that will be called next.

You can retrieve the method arguments from the context.

Calling next executes the remaining filters in the pipeline.

NOTE The `EndpointFilterDelegate` is a named delegate type. It's effectively a `Func<EndpointFilterInvocationContext, ValueTask<object?>>`.

There are many parallels between the middleware pipeline and the filter endpoint pipeline, and we'll explore them in section 5.4.1.

5.4.1 Adding multiple filters to an endpoint

The middleware pipeline is typically the best place for handling cross-cutting concerns such as logging, authentication, and authorization, as these functions apply to all requests. Nevertheless, it can be common to have additional cross-cutting concerns that are endpoint-specific, as we've already discussed. If you need many endpoint-specific operations, you might consider using multiple endpoint filters.

As you saw in figure 5.8, adding multiple filters to an endpoint builds up a pipeline. Like the middleware pipeline, the endpoint filter pipeline can execute code both before and after the rest of the pipeline executes. Similarly, the filter pipeline can short-circuit in the same way as the middleware pipeline by returning a result and not calling `next`.

> **NOTE** You've already seen an example of a short circuit in the filter pipeline. In listing 5.9 we short-circuit the pipeline if the `id` is invalid by returning a Problem Details object instead of calling `next(context)`.

As with middleware, the order in which you add filters to the endpoint filter pipeline is important. The filters you add first are called first in the pipeline, and filters you add last are called last. On the return journey through the pipeline, after the endpoint handler is invoked, the filters are called in reverse order, as with the middleware pipeline. As an example, consider the following listing, which adds an extra filter to the endpoint shown in listing 5.9.

Listing 5.10 Adding multiple filters to the endpoint filter pipeline

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter(ValidationHelper.ValidateId)
    .AddEndpointFilter(async (context, next) =>
    {
        app.Logger.LogInformation("Executing filter...");
        object? result = await next(context);
        app.Logger.LogInformation($"Handler result: {result}");
        return result;
    });

app.Run();
```

Adds a new filter using a lambda function

Adds the validation filter as before

Logs a message before executing the rest of the pipeline

Executes the remainder of the pipeline and the endpoint handler

Returns the result unmodified

Logs the result returned by the rest of the pipeline

The extra filter is implemented as a lambda function and simply writes a log message when it executes. Then it runs the rest of the filter pipeline (which contains only the endpoint handler in this example) and logs the result returned by the pipeline. Chapter 26 covers logging in detail. For this example, we'll look at the logs written to the console.

Figure 5.9 shows the log messages written when we send two requests to the API in listing 5.10. The first request is for an entry that exists, so it returns a `200 OK` result. The second request uses an invalid `id` format, so the first filter rejects it. Figure 5.9 shows that neither the second filter nor the endpoint handler runs in this case; the filter pipeline has been short-circuited.

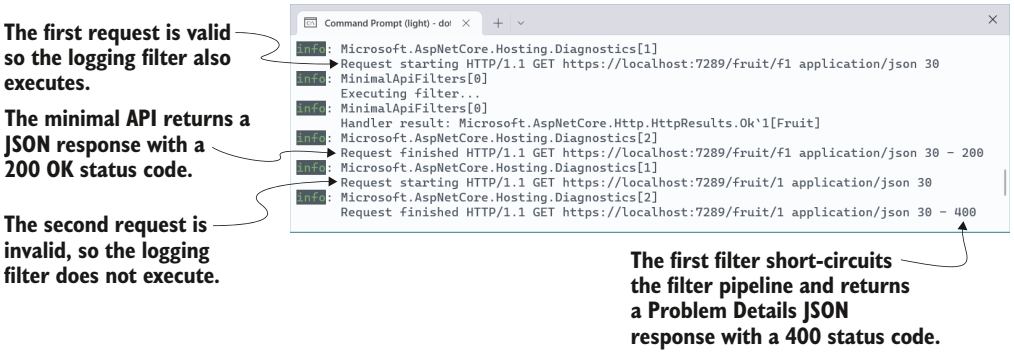


Figure 5.9 Sending two requests to the API from listing 5.10. The first request is valid, so both filters execute. An invalid `id` is provided in the second request, so the first filter short-circuits the requests, and the second filter doesn't execute.

By adding calls to `AddEndpointFilter`, you can create arbitrarily large endpoint filter pipelines, but the fact that you can doesn't mean you should. Moving code to filters can reduce clutter in your endpoints, but it makes the flow of your application harder to understand. I suggest that you avoid using filters unless you find duplicated code in multiple endpoints, and then favor a filter over a simple method call only if it significantly simplifies the code required.

5.4.2 Filters or middleware: Which should you choose?

The endpoint filter pipeline is similar to the middleware pipeline in many ways, but you should consider several subtle differences when deciding which approach to use. The similarities include three main parallels:

- *Requests pass through a middleware component on the way in, and responses pass through again on the way out.* Similarly, endpoint filters can run code before calling the next filter in the pipeline and can run code after the response is generated, as shown in figure 5.8.
- *Middleware can short-circuit a request by returning a response instead of passing it on to later middleware.* Filters can also short-circuit the filter pipeline by returning a response.
- *Middleware is often used for cross-cutting application concerns, such as logging, performance profiling, and exception handling.* Filters also lend themselves to cross-cutting concerns.

By contrast, there are three main differences between middleware and filters:

- Middleware can run for all requests; filters will run only for requests that reach the `EndpointMiddleware` and execute the associated endpoint.
- Filters have access to additional details about the endpoint that will execute, such as the return value of the endpoint, for example an `IResult`. Middleware in general won't see these intermediate steps, so it sees only the generated response.
- Filters can easily be restricted to a subset of requests, such as a single endpoint or a group of endpoints. Middleware generally applies to all requests (though you can achieve something similar with custom middleware components).

That's all well and good, but how should we interpret these differences? When should we choose one over the other?

I like to think of middleware versus filters as a question of specificity. Middleware is the more general concept, operating on lower-level primitives such as `HttpContext`, so it has wider reach. If the functionality you need has no endpoint-specific requirements, you should use a middleware component. Exception handling is a great example; exceptions could happen anywhere in your application, and you need to handle them, so using exception-handling middleware makes sense.

On the other hand, if you *do* need access to endpoint details, or if you want to behave differently for some requests, you should consider using a filter. Validation is a good example. Not all requests need the same validation. Requests for static files, for example, don't need parameter validation, the way requests to an API endpoint do. Applying validation to the endpoints via filters makes sense in this case.

> **TIP** Where possible, consider using middleware for cross-cutting concerns. Use filters when you need different behavior for different endpoints or where the functionality relies on endpoint concepts such as `IResult` objects.

So far, the filters we've looked at have been specific to a single endpoint. In section 5.4.3 we look at creating generic filters that you can apply to multiple endpoints.

5.4.3 Generalizing your endpoint filters

One common problem with filters is that they end up closely tied to the *implementation* of your endpoint handlers. Listing 5.9, for example, assumes that the `id` parameter is the first parameter in the method. In this section you'll learn how to create generalized versions of filters that work with *multiple* endpoint handlers.

The `fruit` API we've been working with in this chapter contains several endpoint handlers that take multiple parameters. The `MapPost` handler, for example, takes a `string id` parameter and a `Fruit fruit` parameter:

```
app.MapPost("/fruit/{id}", (string id, Fruit fruit) => { /* */ });
```

In this example, the `id` parameter is listed first, but there's no requirement for that to be the case. The parameters to the handler could be reversed, and the endpoint would be functionally identical:

```
app.MapPost("/fruit/{id}", (Fruit fruit, string id) => { /* */ });
```

Unfortunately, with this order, the `ValidateId` filter described in listing 5.9 won't work. The `ValidateId` filter assumes that the first parameter to the handler is `id`, which isn't the case in our revised `MapPost` implementation.

ASP.NET Core provides a solution that uses a factory pattern for filters. You can register a filter factory by using the `AddEndpointFilterFactory()` method. A *filter factory* is a method that returns a *filter function*. ASP.NET Core executes the filter factory when it's building your app and incorporates the returned filter into the filter pipeline for the app, as shown in figure 5.10. You can use the same filter-factory function to emit a different filter for each endpoint, with each filter tailored to the endpoint's parameters.

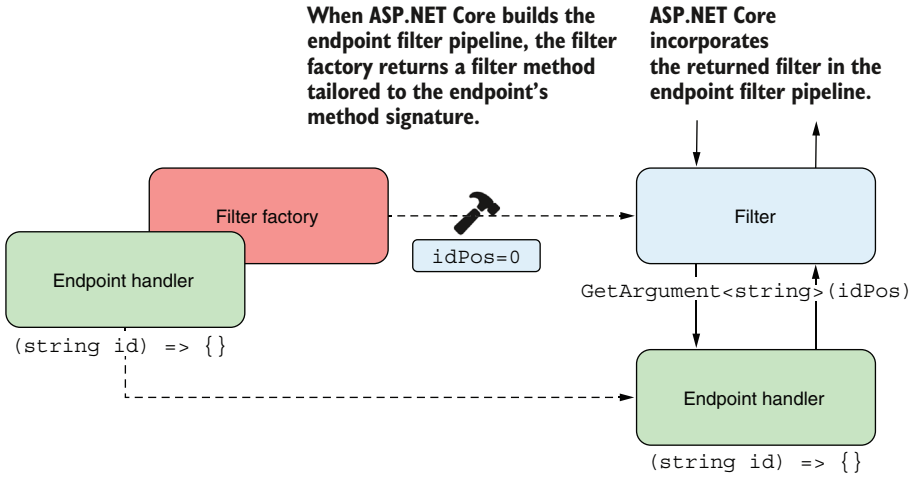


Figure 5.10 A filter factory is a generalized way to add endpoint filters. The factory reads details about the endpoint, such as its method signature, and builds a filter function. This function is incorporated into the final filter pipeline for the endpoint. The build step means that a single filter factory can create filters for multiple endpoints with different method signatures.

Listing 5.11 shows an example of the factory pattern in practice. The filter factory is applied to multiple endpoints. For each endpoint, the factory first checks for a parameter called `id`; if it doesn't exist, the factory returns `next` and doesn't add a filter to the pipeline. If the `id` parameter exists, the factory returns a filter function, which is virtually identical to the filter function in listing 5.9; the main difference is that this filter handles a variable location of the `id` parameter.

Listing 5.11 Using a filter factory to create an endpoint filter

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilterFactory(ValidationHelper.ValidateIdFactory);

app.MapPost("/fruit/{id}", (Fruit fruit, string id) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>
          {
              { "id", new[] { "A fruit with this id already exists" } }
          }))
  .AddEndpointFilterFactory(ValidationHelper.ValidateIdFactory);

app.Run();

class ValidationHelper
{
    internal static EndpointFilterDelegate ValidateIdFactory(
        EndpointFilterFactoryContext context,
        EndpointFilterDelegate next)
    {
        ParameterInfo[] parameters =
            context.MethodInfo.GetParameters();
        int? idPosition = null;
        for (int i = 0; i < parameters.Length; i++)
        {
            if (parameters[i].Name == "id" &&
                parameters[i].ParameterType == typeof(string))
            {
                idPosition = i;
                break;
            }
        }

        if (!idPosition.HasValue)
        {
            return next;
        }

        return async (invocationContext) =>
        {
            var id = invocationContext
                .GetArgument<string>(idPosition.Value);
            if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
            {
                return Results.ValidationProblem(
                    new Dictionary<string, string[]>
```

The filter factory can handle endpoints with different method signatures.

The context parameter provides details about the endpoint handler method.

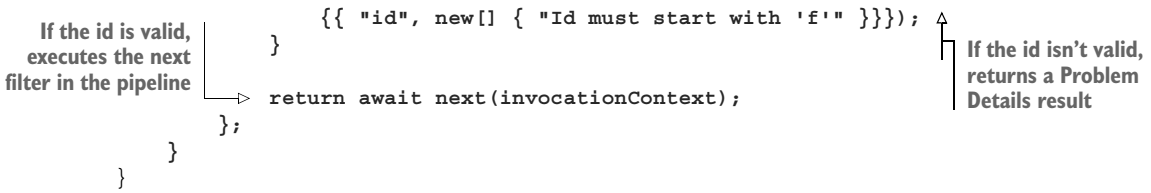
GetParameters() provides details about the parameters of the handler being called.

Loops through the parameters to find the string id parameter and record its position

If the id parameter exists, returns a filter function (the filter executed for the endpoint)

If the id parameter isn't not found, doesn't add a filter, but returns the remainder of the pipeline

If the id isn't valid, returns a Problem Details result




```
                {{ "id", new[] { "Id must start with 'f'" }}});
        }

        return await next(invocationContext);
    };
}
}
```

The code in listing 5.11 is more complex than anything else we've seen so far, as it has an extra layer of abstraction. The endpoint middleware passes an `EndpointFilterFactoryContext` object to the factory function, which contains extra details about the endpoint in comparison to the context passed to a normal filter function. Specifically, it includes a `MethodInfo` property and an `EndpointMetadata` property.

NOTE You'll learn about endpoint metadata in chapter 6.

The `MethodInfo` property can be used to control how the filter is created based on the definition of the endpoint handler. Listing 5.11 shows how you can loop through the parameters to check for the details you need—a `string id` parameter, in this case—and customize the filter function you return.

If you find all these method signatures to be confusing, I don't blame you. Remembering the difference between an `EndpointFilterFactoryContext` and `EndpointFilterInvocationContext` and then trying to satisfy the compiler with your lambda methods can be annoying. Sometimes, you yearn for a good ol' interface to implement. Let's do that now.

5.4.4 Implementing the IEndpointFilter interface

Creating a lambda method for `AddEndpointFilter()` that satisfies the compiler can be a frustrating experience, depending on the level of support your integrated development environment (IDE) provides. In this section you'll learn how to sidestep the issue by defining a class that implements `IEndpointFilter` instead.

You can implement `IEndpointFilter` by defining a class with an `InvokeAsync()` that has the same signature as the lambda defined in listing 5.9. The advantage of using `IEndpointFilter` is that you get IntelliSense and autocompletion for the method signature. The following listing shows how to implement an `IEndpointFilter` class that's equivalent to listing 5.9.

Listing 5.12 Implementing `IEndpointFilter`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
```

```
  .AddEndpointFilter<IdValidationFilter>();

app.Run();

class IdValidationFilter : IEndpointFilter
{
    public async ValueTask<object?> InvokeAsync(
        EndpointFilterInvocationContext context,
        EndpointFilterDelegate next)
    {
        var id = context.GetArgument<string>(0);
        if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
        {
            return Results.ValidationProblem(
                new Dictionary<string, string[]>
                {
                    {"id", new[]{"Invalid format. Id must start with 'f'"}}
                });
        }

        return await next(context);
    }
}
```

Adds the filter using the generic AddEndpointFilter method

The filter must implement IEndpointFilter . . .

. . . which requires implementing a single method.

Implementing `IEndpointFilter` is a good option when your filters become more complex, but note that there's no equivalent interface for the filter-factory pattern shown in section 5.4.3. If you want to generalize your filters with a filter factory, you'll have to stick to the lambda (or helper-method) approach shown in listing 5.11.

5.5 Organizing your APIs with route groups

One criticism levied against minimal APIs in .NET 6 was that they were necessarily quite verbose, required a lot of duplicated code, and often led to large endpoint handler methods. .NET 7 introduced two new mechanisms to address these critiques:

- *Filters*—Introduced in section 5.4, filters help separate validation checks and cross-cutting functions such as logging from the important logic in your endpoint handler functions.
- *Route groups*—Described in this section, route groups help reduce duplication by applying filters and routing to multiple handlers at the same time.

When designing APIs, it's important to maintain consistency in the routes you use for your endpoints, which often means duplicating part of the route pattern across multiple APIs. As an example, all the endpoints in the `fruit` API described throughout this chapter (such as in listing 5.3) start with the route prefix `/fruit`:

- `MapGet("/fruit", () => {/* */})`
- `MapGet("/fruit/{id}", (string id) => {/* */})`
- `MapPost("/fruit/{id}", (Fruit fruit, string id) => {/* */})`
- `MapPut("/fruit/{id}", (Fruit fruit, string id) => {/* */})`
- `MapDelete("/fruit/{id}", (string id) => {/* */})`

Additionally, the last four endpoints need to validate the `id` parameter. This validation can be extracted to a helper method and applied as a filter, but you still need to *remember* to apply the filter when you add a new endpoint.

All this duplication can be removed by using route groups. You can use route groups to extract common path segments or filters to a single location, reducing the duplication in your endpoint definitions. You create a route group by calling `MapGroup("/fruit")` on the `WebApplication` instance, providing a route prefix for the group (`"/fruit"`, in this case), and `MapGroup()` returns a `RouteGroupBuilder`.

When you have a `RouteGroupBuilder`, you can call the same `Map*` extension methods on `RouteGroupBuilder` as you do on `WebApplication`. The only difference is that all the endpoints you define on the group will have the prefix `"/fruit"` applied to each endpoint you define, as shown in figure 5.11. Similarly, you can call `AddEndpointFilter()` on a route group, and all the endpoints on the group will also use the filter.

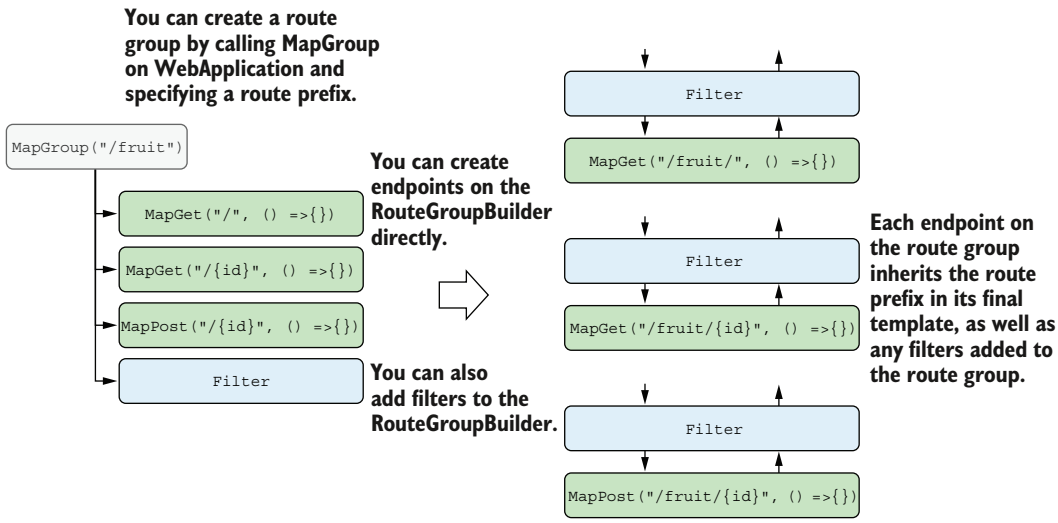


Figure 5.11 Using route groups to simplify the definition of endpoints. You can create a route group by calling `MapGroup()` and providing a prefix. Any endpoints created on the route group inherit the route template prefix, as well as any filters added to the group.

You can even create nested groups by calling `MapGroup()` on a group. The prefixes are applied to your endpoints in order, so the first `MapGroup()` call defines the prefix used at the start of the route. `app.MapGroup("/fruit").MapGroup("/citrus")`, for example, would have the prefix `"/fruit/citrus"`.

> **TIP** If you don't want to add a prefix but still want to use the route group for applying filters, you can pass the prefix `"/"` to `MapGroup()`.

Listing 5.13 shows an example of rewriting the `fruit` API to use route groups. It creates a top-level `fruitApi`, which applies the `"/fruit"` prefix, and creates a nested route

group called `fruitApiWithValidation` for the endpoints that require a filter. You can find the complete example comparing the versions with and without route groups in the source code for this chapter.

Listing 5.13 Reducing duplication with route groups

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

RouteGroupBuilder fruitApi = app.MapGroup("/fruit");

fruitApi.MapGet("/", () => _fruit);

RouteGroupBuilder fruitApiWithValidation = fruitApi.MapGroup("/")
    .AddEndpointFilter(ValidationHelper.ValidateIdFactory);

fruitApiWithValidation.MapGet("/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404));

fruitApiWithValidation.MapPost("/{id}", (Fruit fruit, string id) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>
          {
              { "id", new[] { "A fruit with this id already exists" } }
          }));

fruitApiWithValidation.MapPut("/{id}", (string id, Fruit fruit) =>
{
    _fruit[id] = fruit;
    return Results.NoContent();
});

fruitApiWithValidation.MapDelete("/fruit/{id}", (string id) =>
{
    _fruit.TryRemove(id, out _);
    return Results.NoContent();
});

app.Run();
```

Creates a route group by calling MapGroup and providing a prefix

Endpoints defined on the route group will have the group prefix prepended to the route.

You can create nested route groups with multiple prefixes.

You can add filters to the route group . . .

. . . and the filter will be applied to all the endpoints defined on the route group.

In .NET 6, minimal APIs were a bit too verbose to be generally recommended, but with the addition of route groups and filters, minimal APIs have come into their own. In chapter 6 you'll learn more about routing and route template syntax, as well as how to generate links to other endpoints.

Summary

- HTTP verbs define the semantic expectation for a request. `GET` is used to fetch data, `POST` creates a resource, `PUT` creates or replaces a resource, and `DELETE` removes a resource. Following these conventions will make your API easier to consume.
- Each HTTP response includes a status code. Common codes include `200 OK`, `201 Created`, `400 Bad Request`, and `404 Not Found`. It's important to use the correct status code, as clients use these status codes to infer the behavior of your API.
- An HTTP API exposes methods or endpoints that you can use to access or change data on a server using the HTTP protocol. An HTTP API is typically called by mobile or client-side web applications.
- You define minimal API endpoints by calling `Map*` functions on the `WebApplication` instance, passing in a route pattern to match and a handler function. The handler functions runs in response to matching requests.
- There are different extension methods for each HTTP verb. `MapGet` handles `GET` requests, for example, and `MapPost` maps `POST` requests. You use these extension methods to define how your app handles a given route and HTTP verb.
- You can define your endpoint handlers as lambda expressions, `Func<T, TResult>` and `Action<T>` variables, local functions, instance methods, or static methods. The best approach depends on how complex your handler is, as well as personal preference.
- Returning `void` from your endpoint handler generates a `200` response with no body by default. Returning a string generates a `text/plain` response. Returning an `IResult` instance can generate any response. Any other object returned from your endpoint handler is serialized to JSON. This convention helps keep your endpoint handlers succinct.
- You can customize the response by injecting an `HttpResponse` object into your endpoint handler and then setting the status code and response body. This approach can be useful if you have complex requirements for an endpoint.
- The `Results` and `TypedResults` helpers contain static methods for generating common responses, such as a `404` Not Found response using `Results.NotFound()`. These helpers simplifying returning common status codes.
- You can return a standard Problem Details object by using `Results.Problem()` and `Results.ValiationProblem()`. `Problem()` generates a `500` response by default (which can be changed), and `ValidationProblem()` generates a `400` response, with a list of validation errors. These methods make returning Problem Details objects more concise than generating the response manually.
- You can use helper methods to generate other common result types on `Results`, such as `File()` for returning a file from disk, `Bytes()` for returning arbitrary binary data, and `Stream()` for returning an arbitrary stream.

- You can extract common or tangential code from your endpoint handlers by using endpoint filters, which can keep your endpoint handlers easy to read.
- Add a filter to an endpoint by calling `AddEndpointFilter()` and providing the lambda function to run (or use a static/instance method). You can also implement `IEndpointFilter` and call `AddEndpointFilter<T>()`, where `T` is the name of your implementing class.
- You can generalize your filter functions by creating a factory, using the overload of `AddEndpointFilter()` that takes an `EndpointFilterFactoryContext`. You can use this approach to support endpoint handlers with various method signatures.
- You can reduce duplication in your endpoint routes and filter configuration by using route groups. Call `MapGroup()` on `WebApplication`, and provide a prefix. All endpoints created on the returned `RouteGroupBuilder` will use the prefix in their route templates.
- You can also call `AddEndpointFilter()` on route groups. Any endpoints defined on the group will also have the filter, as though you defined them on the endpoint directly, removing the need to duplicate the call on each endpoint.